**Dissassembly**

000017c5 int32\_t main(int32\_t argc, char\*\* argv, char\*\* envp)

000017f1 int64\_t var\_70 = 0xc

00001808 int64\_t rax\_1 = 0x1c

00001810 int64\_t rdx = 0

0000181c void\* buf = &var\_78 - divu.dp.q(rdx:rax\_1, 0x10) \* 0x10

00001835 printf(format: "Enter your secret: ", argv, modu.dp.q(rdx:rax\_1, 0x10), 0x10)

00001852 fgets(buf, n: 0xd, fp: \_\_TMC\_END\_\_)

0000186c int64\_t var\_60 = 0x54

000018b6 void\* rsp = buf - divu.dp.q(0:0x64, 0x10) \* 0x10

000018c7 gen\_padding(rsp)

000018da int64\_t var\_50 = 0x60

00001905 void\* rsp\_1 = rsp - divu.dp.q(0:0x70, 0x10) \* 0x10

0000191d strcpy(rsp\_1, buf)

00001930 strcat(rsp\_1, rsp)

00001943 int64\_t var\_40 = 0x60

0000196e void\* rsp\_2 = rsp\_1 - divu.dp.q(0:0x70, 0x10) \* 0x10

00001986 shuffle(rsp\_1, rsp\_2)

00001999 int64\_t var\_30 = 0xc

000019c4 void\* rsp\_3 = rsp\_2 - divu.dp.q(0:0x1c, 0x10) \* 0x10

000019dc fold(rsp\_2, rsp\_3)

000019f7 printf(format: "Your processed secret: %s\n", rsp\_3)

00001189 int64\_t gen\_padding(int64\_t arg1)

00001195 void\* fsbase

00001195 int64\_t rax = \*(fsbase + 0x28)

000011ae int64\_t var\_1d

000011ae \_\_builtin\_strcpy(dest: &var\_1d, src: "jvucsiwfaebq")

000011ae

00001255 for (int32\_t i = 0; i s<= 6; i += 1)

0000124b for (int32\_t j = 0; j s< 0xc; j += 1)

0000123d \*(sx.q(j + 0xc \* i) + arg1) = ((sx.d(\*(&var\_1d + sx.q(j))) - 0x61) \* j \* (i + 3) s% 0x1a).b + 0x61

00001272 char\* shuffle(void\* arg1, void\* arg2)

0000127e int32\_t var\_c = 0

0000127e

000012ca for (int32\_t i = 0; i s< 0x60; i += 1)

000012bc \*(arg2 + sx.q(i)) = \*(arg1 + sx.q(mods.dp.d(sx.q(i \* 0x11), 0x60)))

000012bc

000012db \*(arg2 + 0x60) = 0

000012e0 return arg2 + 0x60

000012e1 int64\_t fold(char\* arg1, void\* arg2)

000012fb int64\_t rax = \*(fsbase + 0x28)

0000131e int32\_t var\_ac = 0x30

00001333 int32\_t var\_a8 = 0x18

00001348 int32\_t var\_a4 = 0xc

0000135b int64\_t var\_a0 = 0x2f

00001381 void\* rsp = &var\_c8 - divu.dp.q(0:0x3f, 0x10) \* 0x10

0000139b int64\_t var\_90 = 0x2f

000013c1 void\* rsp\_1 = rsp - divu.dp.q(0:0x3f, 0x10) \* 0x10

000013d8 int64\_t var\_80 = 0x2f

000013fb void\* rsp\_2 = rsp\_1 - divu.dp.q(0:0x3f, 0x10) \* 0x10

00001422 strncpy(rsp, arg1, 0x30)

0000144d strncpy(rsp\_1, &arg1[sx.q(var\_ac)], sx.q(var\_ac))

0000144d

000014d7 for (int32\_t i = 0; i s< var\_ac; i += 1)

000014c1 \*(rsp\_2 + sx.q(i)) = ((sx.d(\*(rsp\_1 + sx.q(i))) - 0x61 + sx.d(\*(rsp + sx.q(i))) - 0x61) s% 0x1a).b + 0x61

000014c1

000014e6 int64\_t var\_70 = sx.q(var\_a8) - 1

00001509 void\* rsp\_3 = rsp\_2 - divu.dp.q(0:(sx.q(var\_a8) + 0xf), 0x10) \* 0x10

00001520 int64\_t var\_60 = sx.q(var\_a8) - 1

00001543 void\* rsp\_4 = rsp\_3 - divu.dp.q(0:(sx.q(var\_a8) + 0xf), 0x10) \* 0x10

00001549 char\* var\_58 = rsp\_4

0000155a int64\_t var\_50 = sx.q(var\_a8) - 1

0000157d void\* rsp\_5 = rsp\_4 - divu.dp.q(0:(sx.q(var\_a8) + 0xf), 0x10) \* 0x10

0000159e strncpy(rsp\_3, rsp\_2, sx.q(var\_a8))

000015c6 strncpy(var\_58, sx.q(var\_a8) + rsp\_2, sx.q(var\_a8))

000015c6

0000164d for (int32\_t i\_1 = 0; i\_1 s< var\_a8; i\_1 += 1)

00001637 \*(rsp\_5 + sx.q(i\_1)) = ((sx.d(var\_58[sx.q(i\_1)]) - 0x61 + sx.d(\*(rsp\_3 + sx.q(i\_1))) - 0x61) s% 0x1a).b + 0x61

00001637

0000165c int64\_t var\_40 = sx.q(var\_a4) - 1

0000167f void\* rsp\_6 = rsp\_5 - divu.dp.q(0:(sx.q(var\_a4) + 0xf), 0x10) \* 0x10

00001696 int64\_t var\_30 = sx.q(var\_a4) - 1

000016bf char\* var\_28 = rsp\_6 - divu.dp.q(0:(sx.q(var\_a4) + 0xf), 0x10) \* 0x10

000016da strncpy(rsp\_6, rsp\_5, sx.q(var\_a4))

00001702 strncpy(var\_28, sx.q(var\_a4) + rsp\_5, sx.q(var\_a4))

00001702

0000178f for (int32\_t i\_2 = 0; i\_2 s< var\_a4; i\_2 += 1)

0000177a \*(var\_c8 + sx.q(i\_2)) = ((sx.d(var\_28[sx.q(i\_2)]) - 0x61 + sx.d(\*(rsp\_6 + sx.q(i\_2))) - 0x61) s% 0x1a).b + 0x61

0000177a

000017a4 \*(var\_c8 + sx.q(var\_a4)) = 0

000017ba \_\_stack\_chk\_fail()

000017ba noreturn

Copy

The binary contains a few key functions:

* **generate\_padding()**: produces a fixed 84-character padding string from a hardcoded source "jvucsiwfaebq".
* **shuffle(buffer)**: reorders a 96-character buffer based on some indices, call that **SHUFFLE\_INDICES**.
* **fold(buffer)**: compresses the buffer from 96 → 48 → 24 → 12 characters using mod-26 character addition.
* **main(input)**: applies padding, shuffle, then fold to return a final 12-character string.

It's trivial to attempt a brute force. But, the brute-force method is clearly inefficient, so we analyze and reverse the operations.

**Understanding Shuffle**

**SHUFFLE\_INDICES[i] = (i \* 17) % 96** for **i** in **0..95**

Since 17 is coprime to 96, the mapping is bijective and thus reversible. To invert:

reverse\_shuffle[i] = j # where SHUFFLE\_INDICES[j] = i

# that is:

reverse\_shuffle[SHUFFLE\_INDICES[i]] = i

Copy

More compactly, since 17 \* 17 ≡ 1 mod 96, the modular inverse of 17 mod 96 is 17. So to reverse:

original[j] = shuffled[(j \* 17) % 96]

Copy

This lets us reconstruct the original input+padding buffer from a shuffled one.

**Understanding Fold**

Folding is a 3-step reduction:

1. Fold 96 → 48: **folded[i] = (ord(S[i]) + ord(S[i+48]) - 194) % 26 + 97**
2. Fold 48 → 24: same pattern on result
3. Fold 24 → 12: same again to produce final string

Each final output character is a linear combination of 8 shuffled characters:

fold3[m] = (S[m] + S[m+12] + S[m+24] + S[m+36] + S[m+48] + S[m+60] + S[m+72] + S[m+84]) % 26

Copy

Each **S[i]** maps to either:

* An input character (if SHUFFLE\_INDICES[i] < 12), or
* A known padding character (SHUFFLE\_INDICES[i] ≥ 12 → padding[SHUFFLE\_INDICES[i] - 12])

This allows us to express each fold3[i] (i.e., target character) as a linear sum of one unknown and 7 knowns.

**Solution Strategy**

1. For each i in 0..11 (target character):
   * Identify the 8 contributing positions in the shuffled buffer: i, i+12, ..., i+84
   * For each, map SHUFFLE\_INDICES[pos]
   * If SHUFFLE\_INDICES[pos] < 12 → input[SHUFFLE\_INDICES[pos]] is unknown
   * Otherwise → add padding value to sum
2. Each target character gives a direct equation: x\_k = (target\_val - sum\_of\_padding\_contributions) % 26
3. Construct input string from x\_0 .. x\_11

**Solution Code**

import string

# Precomputed padding generation

def generate\_padding():

src = "jvucsiwfaebq"

padding = []

for i in range(7): # 0-6 inclusive (7 rows)

for j in range(12):

c = ord(src[j]) - ord('a')

val = (c \* j \* (i + 3)) % 26

padding.append(chr(val + ord('a')))

return ''.join(padding)

PADDING = generate\_padding()

SHUFFLE\_INDICES = [(i \* 17) % 96 for i in range(96)]

def compute\_input\_vars(target):

input\_chars = [None] \* 12

target\_nums = [ord(c) - ord('a') for c in target]

for m in range(12):

sum\_padding = 0

input\_idx = None

for k in range(8):

i = m + 12 \* k

buffer\_idx = SHUFFLE\_INDICES[i]

if buffer\_idx < 12:

input\_idx = buffer\_idx

else:

pad\_val = ord(PADDING[buffer\_idx - 12]) - ord('a')

sum\_padding += pad\_val

if input\_idx is None:

raise Exception("No input variable found")

input\_val = (target\_nums[m] - sum\_padding) % 26

input\_chars[input\_idx] = chr(input\_val + ord('a'))

return ''.join(input\_chars)

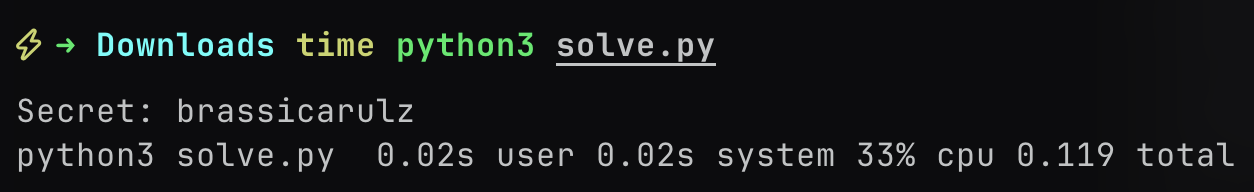
TARGET = "bypkrpihayqo"

secret = compute\_input\_vars(TARGET)

print("Secret:", secret)

**Result**

Running the above returns the 12-character secret input that produces the encrypted target "brassicarulz".

*it's sure a hell of a lot faster than the probably 500 years needed to brute-force LMAO*

No brute-force required; we reverse-engineered the constant-folded structure mathematically.

**flag is:** **sdctf{brassicarulz}**